Rapport de projet POOIG

TEIXEIRA MEDEIROS, Claudio

MAUDET, Benjamin

Binôme 13

I) Introduction

“The Alchemist” est le jeu final rendu pour ce projet, clone de Pet Rescue Saga.

Dans ce rendu nous discuterons comment et avec quels outils nous avons développé ce projet. Il sera suivi d’une modélisation graphique UML du projet pour ensuite discuter des principes de Programmation Orientée Objet que nous avons voulu implémenter.

Nous aborderons chacun des conseils donnés pour l’implémentation de ce jeu en comparaison avec ce que nous avons fait. Finalement, nous parlerons des difficultés que nous avons rencontrées, des éventuels problèmes non résolus et les pistes de fonctionnalités qui auraient pu être implémentées.

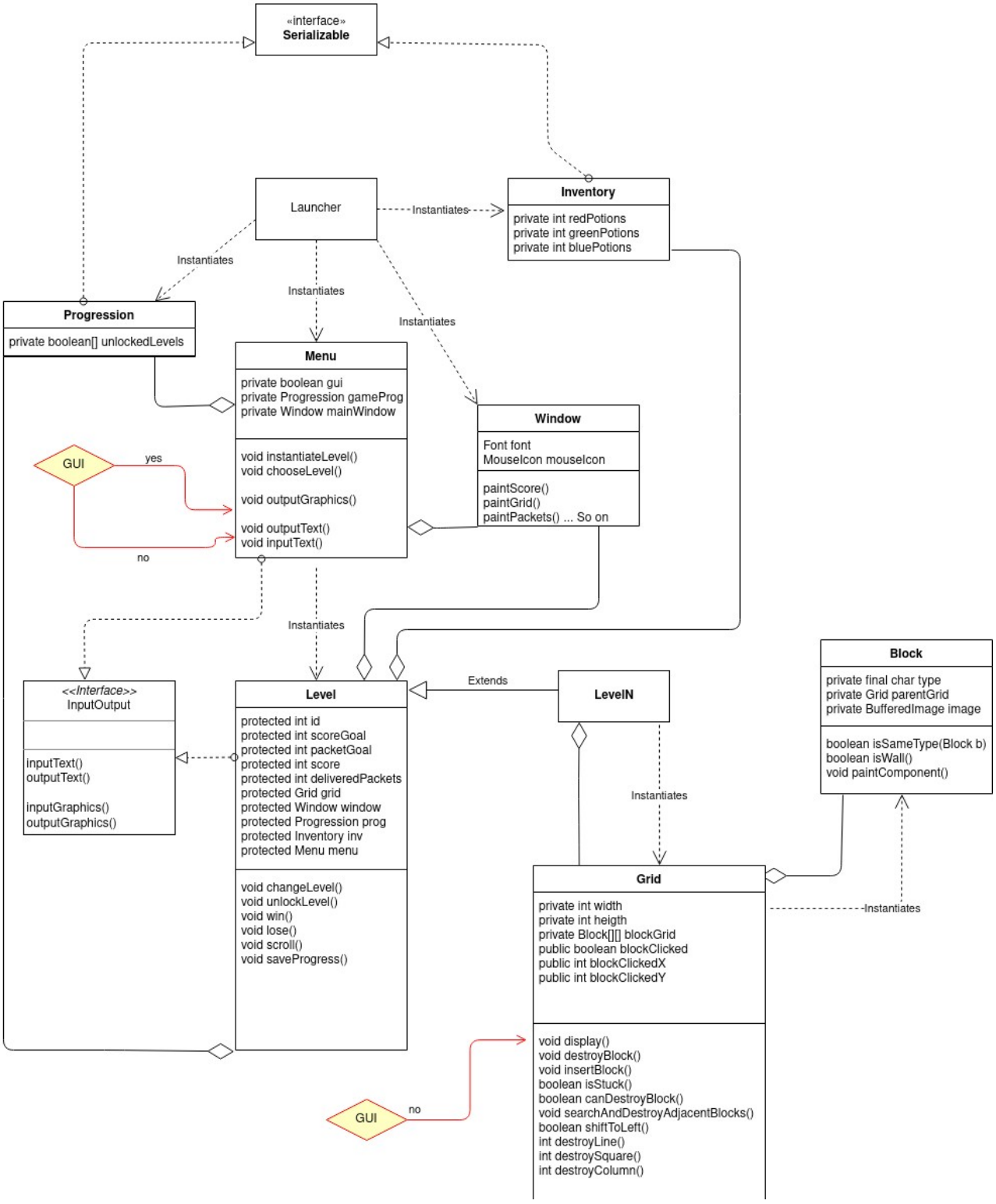
II) Outils

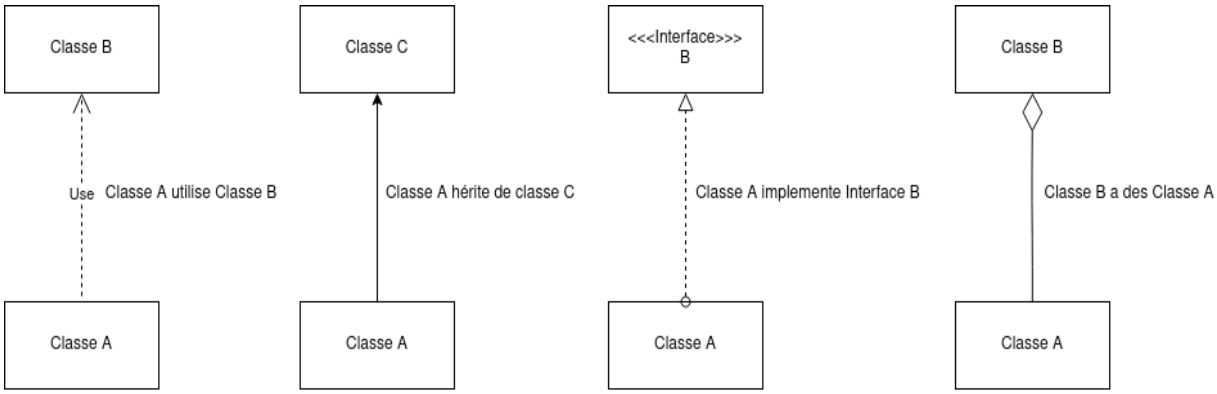
Nous avons travaillé ensemble sur un dépôt GitHub et en communicant via Discord comme se rencontrer en personne était impossible.

Le jeu comporte sons, musiques, images de fond et sprites qui proviennent de plusieurs sources libres de droits et sans redevance. Nous avons utilisé Swing pour les sons et les images.

La modélisation UML a été faite avec le logiciel open-source Drawio (https://github.com/jgraph/drawio-desktop).

III) UML


«interface»
Serializable
Launcher
Instantiates
Inventory
private int redPotions
private int greenPotions
private int bluePotions
Instantiates
Instantiates
Instantiates
Progression
private boolean[] unlockedLevels
Menu
private boolean gui
private Progression gameProg
private Window mainWindow
void instantiateLevel()
void chooseLevel()
void outputGraphics()
void outputText()
void inputText()
Window
Font font
MouseIcon mouseIcon
paintScore()
paintGrid()
paintPackets() ... So on
GUI
yes
no
Instantiates
Block
private final char type
private Grid parentGrid
private BufferedImage image
boolean isSameType(Block b)
boolean isWall()
void paintComponent()
<<Interface>>
InputOutput
inputText()
outputText()
inputGraphics()
outputGraphics()
Level
protected int id
protected int scoreGoal
protected int packetGoal
protected int score
protected int deliveredPackets
protected Grid grid
protected Window window
protected Progression prog
protected Inventory inv
protected Menu menu
void changeLevel()
void unlockLevel()
void win()
void lose()
void scroll()
void saveProgress()
Extends
LevelN
Instantiates
Grid
private int width
private int heigth
private Block[][] blockGrid
public boolean blockClicked
public int blockClickedX
public int blockClickedY
Instantiates
void display()
void destroyBlock()
void insertBlock()
boolean isStuck()
boolean canDestroyBlock()
void searchAndDestroyAdjacentBlocks()
boolean shiftToLeft()
int destroyLine()
int destroySquare()
int destroyColumn()
GUI
no



Vous remarquerez que les classes **AudioManager**, **SpriteManager**, **FontManager**, **CustomButton** et **CustomPopup** ne sont pas présentes. Cela est du au fait qu'elles servent une fonction purement esthétique, de performance ou de refactorisation du code et ne sont pas utiles à notre discussion sur les principes de développement.

LevelN correspond à Level1, Level2, Level3... etc.

III) Principes

Nous n'avons pas suivi un modèle strict **MVC**. Nous comprenons l'avantage de celui-ci dans le cas d'une application : plusieurs vues et contrôleurs peuvent utiliser un seul et même modèle de données pour leurs opérations. Si le modèle est suffisamment bien conçu, on peut même faire varier la source des données alimentant le modèle en changeant de base de données.

Nous trouvions ce modèle moins flexible que ce qu'on pourrait atteindre autrement dans le cadre d'un jeu match 3.

Nous avons décidé que ce jeu contiendrait deux classes principales Menu et Level. Menu permet tout simplement d'instancier les Levels avec les objets nécessaires.

Mais nous étions conscients que pour créer un code propre et modulable il faudrait implémenter les principes de programmation. Nous avons pensé aux principes **SOLID** et retenu principalement :

- **OCP (Open-Closed principle) -** Une entité est ouverte à l'extension, mais fermée à la modification

- **SRP (Single Responsibility principle) -** Une classe ne devrait avoir une responsabilité que sur une seule partie de la fonctionnalité de ce programme, les fonctions et méthodes s'alignent avec cette responsabilité.

- **DIP (Dependency Inversion principle) -** Les modules de haut niveau ne dépendent pas des modules de bas niveau, les deux dépendent des abstractions. Et les abstractions ne dépendent pas des détails, les détails (implémentations concrètes) dépendent des abstractions.

Pour Level nous avons donc choisi de nous inspirer de la stratégie "**Factory method**". Nous avons créé des objets en appelant un **Factory Method** (dans ce cas, **instantiateLevel()** dans Menu) qui utilise le principe de réflexion pour créer des LevelN qui implémentent la classe abstraite Level. Cela nous permet de découpler l'appel de son implémentation (Menu de LevelN) et nous permet d'adhérer au principe **OCP** en rendant la création de niveaux plus flexible et facile.

Pour respecter la **SRP,** chaque classe a sa propre responsabilité. Ses méthodes et attributs restent bien alignés avec ses responsabilités.

- **Launcher** – Permet d'initialiser le jeu.
- **Progression et Inventory** – Permettent de stocker les données nécessaires au jeu.
- **Menu** – Permet à l'utilisateur d'instancier les niveaux.
- **Level** – Mets en place les conditions de jeu et de victoire de chaque niveau.
- **Grid** – Contient le plateau, encapsule la logique de fonctionnement du jeu.
- **Block** – Contient les informations de chaque Block
- **Window** – Permet d'implémenter une vue graphique facilement.

Les modules Menu et Level dépendent de l'abstraction **InputOutput.** Grâce à cette interface, nous pouvons inverser leur dépendances dans le gros de leur fonctionnement : leur affichage et leur réponse aux actions de l'utilisateur. Nous avons fait ça en espérant que cela rende le code plus facilement modulable.

Tout LevelN dépend de l'abstraction **Level**. Les implémentations concrètes dépendent de cette classe abstraite qui est bien définie et permet une grande variété de niveaux différents.

C'est en implémentant ces deux abstractions que nous avions espéré adhérer au principe **DIP.**

IV) Conseils du sujet

Environnement de jeu

Nous avons créé les classes sérialisables où se trouvent les différentes données de progression des joueurs. Ces données sont initialisées une fois par Launcher et ensuite traitées uniquement par Level. Ceci rentre dans les conditions de jeu : la seule façon de progresser est de terminer un niveau.

Pour l'accueil du joueur, nous avons décidé de le déléguer à la classe Menu.

Le plateau

Celui-ci correspond à la classe Grid. Comme susmentionné, elle encapsule la logique du jeu.

Les phases du jeu en lui-même

Au-delà des principes SOLID, nous avons essayé d'écrire des fonctions simples, avec des noms explicites, qui se suivent en ordre d'abstraction pour obtenir une lecture facilité. Peu de blocs de code font plus de 20 lignes, sauf quand il s'agit de création de composants Swing a ajouter a la fenêtre.

Nature des joueurs

Aucun joueur robot ou système d'indices n'ont été implémentés.

Vue et Modèle

Traité dans "Principes".

V) Difficultés

- Malgré notre détachement de MVC, nous avons essayé de garder un maximum la Vue séparée du reste du code, mais des actions de contrôleurs et vue se retrouvent souvent dans une même classe, même si les actions de vue sont toujours englobées par une interface.

- Nous n'avons pas exploité les Workers et les fonctions de threading de Swing de façon efficace.

- Nous n'avions jamais adhéré à des principes de programmation dans nos projets antérieurs. La compréhension et l'adhésion à ces principes lors du design a été particulièrement difficile.

- Certains Events en Swing n'étaient pas particulièrement bien gérés.

VI) Fonctionnalités à implémenter

- Un éditeur de niveaux serait relativement facile à implémenter : l'utilisateur créerait un Grid et une limite de score qu'on passerait à un nouveau fichier .java étendant Level. Il serait nécessaire de faire quelques modifications à Menu pour l'affichage des "niveaux custom".

- Il serait intéressant pour la jouabilité d'ajouter un moyen d'obtenir des potions : un niveau spécial de recharge ou alors un certain score qui donnerait des potions bonus.

- Implémenter des highscores locaux et un système d'étoile comme celui du jeu original.

- Ajouter plus de niveaux, ce qui veut aussi dire plus d'images de fond et plus de musiques, même si cela rentre moins dans le cadre du projet.